Why I Started with Open Source

Open source contribution has always intrigued me. The idea of collaborating with developers worldwide, learning from real-world codebases, and making a tangible impact on software used by millions is incredibly motivating.

My goals were:

- Learn industry-standard development practices
- Connect with the developer community
- Build a portfolio of real contributions
- Understand collaborative software development
- Gain practical experience beyond tutorials

Setting Up My First Project: Nextcloud

Why Nextcloud?

I chose Nextcloud as my first major project because:

1. Well-documented - Excellent documentation for new contributors
2. Active community - Responsive maintainers and helpful community
3. Real-world impact - Used by millions for file storage and collaboration
4. Modern tech stack - PHP, JavaScript, Vue.js
5. Beginner-friendly issues - Tagged issues for first-time contributors

Setting Up the Development Environment

Challenges I Faced

Challenge 1: Large Codebase

- Problem: Nextcloud is a massive project with thousands of files
- Solution: Started by reading the CONTRIBUTING.md file and focused on one module at a time

Challenge 2: Development Environment

```
# Clone the repository
git clone https://github.com/nextcloud/server.git
cd server
# Install dependencies
composer install
npm install
# Set up local development server
php -S localhost:8080
```

- Problem: Setting up all dependencies was complex

- Solution: Used Docker containers to simplify the setup process

Challenge 3: Understanding the Architecture

- Problem: Didn't know where to start making changes
- Solution: Read existing PRs and issues to understand common patterns

Understanding and Choosing a License

One crucial aspect of my project was understanding open source licenses. This was eye-opening!

Why Licenses Matter

A license:

- Protects your work
- Defines how others can use your code
- Clarifies contribution terms
- Prevents legal issues

Licenses I Considered

License	Pros	Cons	Best For
MIT	Simple, permissive, widely used	Minimal protection	Most projects
GPL v3	Strong copyleft, ensures freedom	Restrictive for commercial use	Community-focused projects
Apache 2.0	Patent protection, permissive	More complex	Corporate-backed projects
BSD	Very permissive	Minimal contributor protection	Academic projects

My Choice: MIT License

I chose the MIT License for my project because:

1. Simplicity - Easy to understand for contributors
2. Flexibility - Allows both open source and commercial use
3. Wide adoption - Most developers are familiar with it
4. Minimal restrictions - Encourages maximum collaboration

Pro Tip: Add a LICENSE file to your repository root. GitHub will automatically detect it!

My First Pull Requests

PR #1: Documentation Fix (MERGED ✓)

Repository: Nextcloud Documentation
Issue: Outdated installation instructions
Changes: Updated PHP version requirements and fixed broken links

What I learned:

- Always reference the issue number
- Test all links before submitting
- Clear commit messages are crucial

PR #2: Bug Fix in User Management (MERGED ✓)

Repository: Nextcloud Server
Issue: User search not working with special characters
Changes: Added proper escaping for search queries

```
// Before
$query = "SELECT * FROM users WHERE username LIKE '%$search%'";
// After
$query = $this->db->prepare(
    "SELECT * FROM users WHERE username LIKE :search"
);
$query->execute(['search' => '%' . $this->db->escape($search) . '%']);
```

What I learned:

- Security matters - always sanitize inputs
- Write tests for your changes
- Follow the project's coding standards

PR #3: UI Enhancement (UNDER REVIEW 👀)

Repository: Nextcloud Server
Issue: Improve accessibility of file sharing dialog
Changes: Added ARIA labels and keyboard navigation

What I learned:

- Patience is key - reviews take time
- Be open to feedback and suggestions
- Iterate based on maintainer comments

Contributing to Practice Repositories

To build confidence, I also contributed to beginner-friendly practice repositories:

1. first-contributions

- Task: Add my name to contributors list
- Status: MERGED ✓
- Time to merge: 2 days
- Takeaway: Perfect for learning the fork-clone-PR workflow!

2. HyunCafe/contribute-practice

- Task: Add name and inspirational quote
- Status: MERGED ✓
- Time to merge: 3 days
- Takeaway: Great for understanding markdown formatting and git basics.

3. EddieHubCommunity/hacktoberfest-practice

- Task: Add GitHub profile to community list
- Status: MERGED ✓
- Time to merge: 1 day
- Takeaway: Active community with fast review cycles - highly recommended!

Key Lessons Learned

1. Read CONTRIBUTING.md First

Every project has guidelines. Reading them saves time and shows respect to maintainers.

2. Start Small

My first PRs were:

- Documentation fixes
- Typo corrections
- Adding comments
- Small bug fixes

Don't try to refactor the entire codebase on day one!

3. Communication is Key

Before working on an issue:

1. Comment on the issue expressing interest

2. Ask for clarification if needed
3. Propose your approach
4. Wait for maintainer approval
5. Then start coding

4. Write Clean Commits

Bad commit:

```
git commit -m "fixed stuff"
```

Good commit:

```
git commit -m "fix: Resolve user search issue with special characters
- Added proper input sanitization
- Updated tests for edge cases
- Fixed SQL injection vulnerability
Fixes: #12345"
```

5. Be Patient with Reviews

Timeline	What to Expect
0-2 days	Automated checks run
2-7 days	Initial maintainer review
1-2 weeks	Back-and-forth on changes
2-4 weeks	Final approval and merge

Remember: Maintainers are volunteers with day jobs!

6. Handle Rejection Gracefully

Not all PRs get merged, and that's okay!
When your PR is rejected:

- Thank the reviewer for their time
- Ask for feedback on what to improve
- Learn from the experience
- Move on to the next contribution

7. Test Everything

Before submitting:

```
# Run tests
npm test
# Check code style
npm run lint
# Build the project
npm run build
# Test manually in browser
```

Tips for Future Contributors

Getting Started Checklist

- [ ] Create a GitHub account
- [ ] Set up Git on your computer
- [ ] Learn basic Git commands (clone, branch, commit, push, PR)
- [ ] Find beginner-friendly projects
- [ ] Read project documentation
- [ ] Join project communities (Discord, Slack, Forums)
- [ ] Start with "good first issue" labels
- [ ] Make your first contribution
- [ ] Celebrate!

Finding Projects to Contribute To

Websites:

- Good First Issue
- First Timers Only
- Up For Grabs
- CodeTriage

GitHub Search:

```
label:"good first issue" is:open is:issue language:JavaScript
```

Essential Git Commands

```
# Fork and clone
git clone https://github.com/YOUR_USERNAME/project.git
cd project
# Create a branch
git checkout -b fix-issue-123
# Stage and commit changes
git add .
git commit -m "fix: description of fix"
# Push to your fork
git push origin fix-issue-123
# Update your fork with upstream changes
git remote add upstream https://github.com/ORIGINAL_OWNER/project.git
git fetch upstream
git merge upstream/main
```

Writing Good PRs

Template:

```
## Description
Brief description of what this PR does

## Related Issue
Fixes #123

## Changes Made
- Change 1
- Change 2
- Change 3

## Testing
- [ ] Tested locally
- [ ] Added unit tests
- [ ] Updated documentation

## Screenshots (if applicable)

![ ](https://dev-to-
uploads.s3.amazonaws.com/uploads/articles/q2dj7v3v4s6ix2z0uz57.png)

## Checklist
- [ ] Code follows project style guidelines
- [ ] Self-review of code completed
- [ ] Commented complex code sections
- [ ] Updated documentation
- [ ] No breaking changes
```

My Statistics After 3 Months

Metric	Count
Repositories contributed to	8
Pull requests submitted	15
Pull requests merged	12
Issues opened	5
Comments on discussions	47
Stars received	23

What's Next?

My journey is just beginning! Here's what I'm planning:

1. Contribute to larger projects - Dive deeper into complex issues
2. Become a maintainer - Help review others' PRs
3. Create my own OSS project - Give back to the community
4. Mentor new contributors - Share what I've learned
5. Participate in GSoC - Apply for Google Summer of Code

Final Thoughts

Open source contribution has been one of the most rewarding experiences of my developer journey. It's not just about code – it's about:

- Community - Meeting amazing developers worldwide
- Learning - Exposure to real-world codebases
- Impact - Contributing to software used by millions
- Career - Building a portfolio that speaks for itself
- Growth - Becoming a better developer every day

If you're thinking about starting – just do it! Your first contribution doesn't have to be perfect. The community is welcoming, and everyone started where you are now.

Useful Resources

- GitHub Open Source Guide
- How to Contribute to Open Source
- First Contributions
- The Beginner's Guide to Contributing to Open Source
- Nextcloud Developer Documentation

2. /sbin — System Binaries

- Includes commands used for system administration (requires root access).
- Examples: `mount`, `reboot`, `fdisk`, `deluser`

3. /lib — Shared Libraries

- Stores essential shared libraries and kernel modules used by `/bin` and `/sbin`.
- Think of it as the Linux version of Windows `.dll` files.
- Example: `/lib/x86_64-linux-gnu/libc.so.6`

4. /usr — User Programs and Data

- Contains user-related applications and non-essential binaries.
- Subdirectories:
  - `/usr/bin` → User commands
  - `/usr/sbin` → System binaries
  - `/usr/lib` → Libraries for `/usr/bin` and `/usr/sbin`

5. /usr/local — Locally Installed Software

- Used for software compiled or installed manually by the user.
- Keeps them separate from system-managed packages.
- Example: `/usr/local/bin`, `/usr/local/lib`

6. $PATH — Environment Variable

- Defines directories where the shell looks for executables.
- View it with: `echo $PATH`
- Example: `/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin`

7. which — Locate a Command

- Displays the path of the executable being run.
- Example: `which ls → /bin/ls`

8. /etc — Configuration Files

- Holds system-wide configuration files.
- Examples: `/etc/passwd`, `/etc/hosts`, `/etc/fstab`, `/etc/ssh/sshd_config`

9. /home — User Home Directories

- Each user has their own directory here.
- Examples: `/home/sehran`, `/home/adnan`
- Shortcut: `~` points to your home directory.

10. /boot — Boot Loader Files

- Contains files required to boot the Linux kernel.
- Includes the kernel image, initial RAM disk, and bootloader configurations (like GRUB).
- Examples: `/boot/vmlinuz`, `/boot/initrd.img`, `/boot/grub/grub.cfg`

11. /dev — Device Files

- Represents hardware devices as files.
- Examples:
  - `/dev/sda` → Hard disk
  - `/dev/tty` → Terminal
  - `/dev/null` → Data sink

12. /opt — Optional or Third-Party Software

- Used for optional or vendor-installed applications.
- Example: `/opt/google/chrome/`

13. /var — Variable Data Files

- Contains data that changes frequently, like logs or cache.
- Subdirectories:
  - `/var/log` → System logs
  - `/var/cache` → Cached data
  - `/var/spool` → Pending tasks

14. /tmp — Temporary Files

- Used to store temporary data.
- Files here are usually deleted on reboot.

15. /proc — Process Information

- A virtual filesystem in memory that provides real-time info about the kernel and running processes.
- Examples:
  - `/proc/cpuinfo` → CPU details
  - `/proc/meminfo` → Memory info
  - `/proc/1/` → Process with PID 1

🔥

#node #javascript #beginners

Node.js has become a go-to platform for building fast, scalable, and modern web applications. However, its flexibility can also make it easy to fall into bad habits if you're not careful. Whether you're building a small API or a production-grade system serving millions, following solid best practices can help you write cleaner, faster, and more secure code.

This article covers **essential Node.js best practices** — from structure and performance to security and deployment — based on proven industry standards and real-world experience.

🧱 1. Project Structure and Organization

A clean, modular structure is the foundation of a maintainable Node.js application.

✅ Best Practices

- **Use a layered or MVC structure** (`routes`, `controllers`, `services`, `models`, `middlewares`, `utils`).
- For larger projects, **group by feature or domain** (e.g., `/users`, `/auth`, `/payments`).
- Manage configuration using environment variables (`.env`) and a config loader like `dotenv` or `config`.

Example Folder Structure:

```
src/
 ├── config/
 ├── controllers/
 ├── middlewares/
 ├── models/
 ├── routes/
 ├── services/
 ├── utils/
 └── app.js
```

⛶

🧩 2. Code Quality and Maintainability

Readable and consistent code is key to scaling development as your team grows.

✅ Best Practices

- Use **ESLint** and **Prettier** to enforce code style and formatting.
- Prefer **async/await** over callbacks or `.then()` chains.
- Keep functions **small and focused** on one task.
- Return early to reduce nested conditions.
- Use **TypeScript** for large projects to ensure type safety and better developer experience.
- Add **JSDoc comments** for better readability and IDE support.

⚙️ 3. Configuration and Environment Management

Keeping configuration clean and environment-specific prevents production issues.

✅ Best Practices

- Store secrets in `.env` (never commit it to Git).
- Follow the **12-Factor App** principles.
- Validate environment variables using libraries like `joi` or `envalid`.

Example:

```
import dotenv from "dotenv";
dotenv.config();

export const config = {
  port: process.env.PORT || 3000,
  dbUri: process.env.DATABASE_URL,
};
```

🛡️ 4. Security Best Practices

Security should never be an afterthought in backend development.

✅ Best Practices

- Use **Helmet.js** to set secure HTTP headers.
- Validate and sanitize all input (`express-validator`, `Joi`, `xss-clean`, `express-mongo-sanitize`).
- Implement **rate limiting** (`express-rate-limit`) to prevent brute-force attacks.
- Always use **HTTPS** in production.
- Avoid dangerous functions like `eval()` or dynamic `require()`.
- Keep dependencies updated with `npm audit fix`.

⚡ 5. Performance and Scalability

Node.js is fast by design, but there are ways to make it even more efficient.

✅ Best Practices

- Use **PM2** or Node's **Cluster mode** to leverage all CPU cores.
- Enable **gzip compression** (`compression` middleware).
- Cache frequently accessed data using **Redis** or in-memory caching.
- Avoid blocking the event loop — use asynchronous operations.
- Stream large files instead of loading them fully into memory.
- Use **load balancers** like NGINX or HAProxy in production.
- Optimize database queries and use **connection pooling**.

🧵 6. Error Handling and Logging

Errors happen — what matters is how gracefully you handle them.

✅ Best Practices

- Always handle rejected promises.
- Use a **centralized error handler** in Express.
- Avoid exposing internal errors to clients.
- Use structured loggers like **Winston** or **Pino** for consistent logging.
- Track requests with unique correlation IDs for easier debugging.

Example:

```
export function errorHandler(err, req, res, next) {
  console.error(err.stack);
  res.status(err.status || 500).json({
    success: false,
    message: err.message || "Internal Server Error",
  });
}
```

🔐 7. Authentication and Authorization

Secure user management is the backbone of any serious application.

✅ Best Practices

- Use **JWTs** for stateless authentication.
- Hash passwords using **bcrypt** or **argon2**.
- Validate tokens on every request.
- Implement **role-based access control (RBAC)** or permission-based systems.

- Securely refresh tokens and set short expiration times.

🧪 8. Testing and Continuous Integration

Automated testing ensures your app works as expected after every change.

✅ Best Practices

- Use **Jest**, **Mocha**, or **Vitest** for testing.
- Separate **unit**, **integration**, and **end-to-end** tests.
- Use **supertest** to test HTTP routes.
- Run tests in your CI/CD pipeline (GitHub Actions, GitLab CI, Jenkins).

🧰 9. Dependency and Package Management

Dependencies are both powerful and risky. Manage them carefully.

✅ Best Practices

- Use `npm ci` for reproducible builds.
- Keep dependencies minimal.
- Run `npm audit` regularly to detect vulnerabilities.
- Use **nvm** (Node Version Manager) to control Node versions across environments.

☁️ 10. Deployment and Monitoring

Your production setup matters as much as your code.

✅ Best Practices

- Use **PM2** for process management and zero-downtime deployments.
- Use **Docker** for consistent builds across environments.
- Implement **graceful shutdowns** to avoid data loss on restarts.
- Set up **monitoring and logging** (e.g., Datadog, Sentry, New Relic).
- Track memory leaks and event loop delays.

Example:

```
process.on("SIGTERM", () => {
  console.log("Shutting down...");
  server.close(() => process.exit(0));
});
```